CSCI 365 Problem Set 3: Algebraic Data Types & Polymorphism

due Friday, 9 February 2024

Specification

To receive credit for this problem set:

- You must complete at least 8 out of 12 exercises.
- Any code you write must adhere to the Haskell style guide linked from the course web page.

Haskell code should be submitted in one or more `.hs` files. Written exercises may be submitted either as a PDF, or as comments in one of the `.hs` files.

Algebraic Data Types

Exercise 1 Consider the following algebraic data type:

```
data T where
  X :: Bool -> T
  Y :: T
  Z :: T -> T
```

List at least five different values of type `T`.

Exercise 2 In class, we defined an algebraic data type `Shape` as follows:

```
type Coords = (Double, Double)

data Shape where
  Circle :: Coords -> Double -> Shape
  Rect :: Coords -> Coords -> Shape
```

(a) Write a function

```
perimeter :: Shape -> Double
```

which calculates the perimeter of a shape.

(b) Write a function

```
translateX :: Double -> Shape -> Shape
```

which translates any shape horizontally by the given amount.

(c) Now add a constructor to represent squares, and extend the two previous functions to handle squares as well. Make sure that you design the fields of the square constructor appropriately so that it can *only* represent squares.

Exercise 3 Consider the following grammar for propositional logic expressions (without variables):

$$\varphi ::= T \mid F \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi$$

That is, a propositional logic expression φ can be either a constant T or F (representing true and false, respectively), the negation of an expression, or the conjuction or disjunction of two expressions.

Define a Haskell algebraic data type `Prop` to represent such propositional logic expressions, and implement a function

```
eval :: Prop -> Bool
```

to evaluate propositions according to the usual rules of propositional logic.

Exercise 4 The standard Haskell library defines a data type

```
data Maybe a where
  Nothing :: Maybe a
  Just :: a -> Maybe a
```

It is typically used to model potential failure: a value of type `Maybe a` *might* contain a value of type `a`, or it might be `Nothing`. Implement each of the following functions:

(a) `add :: Maybe Int -> Maybe Int -> Maybe Int`, which tries to add two numbers, but returns `Nothing` if either of its arguments is.

(b) `divide :: Integer -> Integer -> Maybe Integer`, which tries to compute the integer quotient of its two arguments, but fails if the second argument is zero.

(c) `collapse :: Maybe (Maybe a) -> Maybe a`, which returns a value of type `a` (wrapped in `Just`) if at all possible, or `Nothing` otherwise.

Exercise 5 The standard Haskell library also defines a data type

```
data Either a b where
  Left :: a -> Either a b
  Right :: b -> Either a b
```

Implement a function of type

```
(a, Either b c) -> Either (a,b) (a,c)
```

Exercise 6 Using our definition of `List` from class, implement a function of type

```
List (Either a b) -> (List a, List b)
```

which separates out a list of `Either a b` values into a list of all the `Left` values and a list of all the `Right` values.

Trees

For the purposes of this problem set, a *binary tree* containing values of type `a` is defined as being either

- empty; or
- a node containing a value of type `a` and (recursively) two binary trees, referred to as the “left” and “right” subtrees. See the illustration in Figure 1, and an example binary tree in Figure 2.

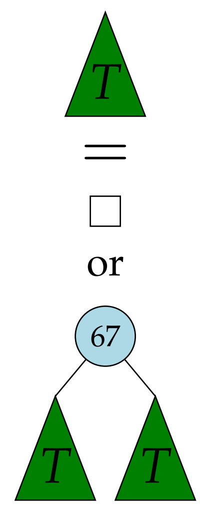

Figure 1: Definition of a binary tree T

Exercise 7 Define a recursive, polymorphic algebraic data type `Tree` which corresponds to the above definition, and define a function

```
incrementTree :: Tree Integer -> Tree Integer
```

which adds one to every `Integer` contained in a tree.

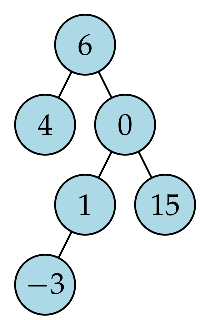

Figure 2: An example binary tree

Exercise 8 Define a function

```
treeSize :: Tree a -> Integer
```

which computes the *size* of a tree, defined as the number of nodes. For example, the tree in Figure 2 has size 6.

A *binary search tree* (BST) is a binary tree of `Integer`s in which the `Integer` value stored in each node is larger than all the `Integer` values in its left subtree, and smaller than all the values in its right subtree. (For the purposes of this problem set, assume that all the values in a binary search tree must be distinct.) For example, the binary tree shown in Figure 2 is not a BST, but the one in Figure 3 is.

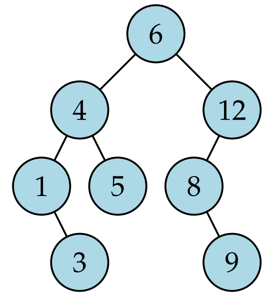

Figure 3: An example binary search tree

Exercise 9 Implement a function

```
bstInsert :: Integer -> Tree Integer -> Tree Integer.
```

Given an integer `i` and a valid BST, `bstInsert` should produce another valid BST which contains `i`. If the input BST already contains `i`, it should be returned unchanged.[1]

[1] It does not matter what `bstInsert` does when given an input `Tree` which is not a valid BST. Later in the course we will talk about ways to use the type system to help enforce invariants such as this.

Exercise 10 Write a function[2]

```
isBST :: Tree Integer -> Bool
```

which checks whether the given `Tree` is a valid BST.

[2] Be careful to check that the value at a node is less than (resp. greater than) *every* node in its left (resp. right) subtree, not just its immediate children. Extra challenge: can you ensure your `isBST` function runs in linear time?

Further Exploration

Exercise 11 Read the article "The Algebra of Algebraic Data Types, Part 1", which can be found at the following URL (the URL should be clickable; if not, try a different PDF reader, copy and paste the URL, or just Google the article's title):

`https://gist.github.com/gregberns/5e9da0c95a9a8d2b6338afe69310b945`

Do the final exercise from the article, that is, show that

```
a -> b -> c === (a,b) -> c
```

in Haskell (corresponding to the algebraic law $(c^b)^a = c^{a \cdot b}$) by writing `to` and `from` functions with appropriate types.

Exercise 12 Read the first page of Philip Wadler's paper *Theorems for free!*,[3] a PDF of which can be found at

`https://dl.acm.org/doi/pdf/10.1145/99370.99404`

I do not expect you to read or understand anything beyond the first page, though you are of course welcome to look if you are curious. Write a 1–2 paragraph response; what you include in your response is up to you, but, for example, you might (but are not required to) include things such as:

- How is this paper related to things we have discussed in class? What are similarities and differences?
- What was going on with Haskell when this paper was published? Who is Phil Wadler and what is his relationship with Haskell and/or functional programming?
- Did you learn anything new, or do you have any new questions sparked by reading this?

[3] Wadler, Philip. Theorems for free! In *Proceedings of the 4th international Conference on Functional Programming Languages and Computer Architecture*, pages 347–359. ACM, 1989.